

LA-UR-19-21951

Approved for public release; distribution is unlimited.

Title: Partial program correctness

Author(s): Herring, Stuart Davis

Intended for: WG21, 2019-02-18/2019-02-23 (Kona, Hawaii, United States)
Web

Issued: 2019-03-05

Disclaimer:

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

P1494R0: Partial program correctness

Audience: EWG; CWG; LEWG; WG14
S. Davis Herring <herring@lanl.gov>
Los Alamos National Laboratory
February 15, 2019

Problem

The termination of the program on violation of a checked contract (with continuation mode “off”) allows additional assumptions on the part of the optimizer. Of course, evaluating the contract condition may more than offset such optimization. Since disabling contract checking is meant to allow faster execution, it seems reasonable to retain the assumptions; [dcl.attr.contract.check]/4 in N4800 therefore makes the behavior of an *unevaluated* check undefined if it “would evaluate to `false`”. Some users consider this introduction of undefined behavior to be an important benefit for optimization.

Others have [expressed concerns](#) over the possibility of adding contracts producing a less reliable program. In particular, unlike the checked termination, the undefined behavior exhibits “time travel” and affects the interpretation of code *preceding* the contract condition. In San Diego, EWG considered multiple presentations on the subject and expressed a preference to remove that effect, partly out of concern over the possibility of undefined behavior *within* a contract check:

```
void f(int *p) [[expects: p]] [[expects: *p<5]];
```

Here, if the violation handler is known to always return (e.g., it merely calls `std::printf`) and the continuation mode is “on”, the first precondition can be *entirely* elided by the implementation: any path of execution either satisfies the condition or has undefined behavior. If it does have undefined behavior, the implementation is completely unconstrained “even with regard to operations preceding the first undefined operation” ([intro.abstract]/5).

Extensive Evolution reflector discussion has continued since, and Kona will see several proposals to alter the treatment of contract checks and the suite of extra-textual controls for them. Many have [wondered](#) about the possibility of making a contract violation handler “opaque to optimization”, so that the first precondition must be checked on the supposition that the handler might not return (but rather throw or terminate). The capability of establishing such a “checkpoint”, where subsequent program behavior, *even if undefined*, does not affect the preceding behavior, would be useful in general for purposes of stability and debugging.

Previous work

D1343R1 suggested making a contract violation itself a side effect, but this does not rise to the level of *observable behavior* ([intro.abstract]/6). Optimization need preserve nothing else; what’s worse, even the observable behavior of a program may be compromised if undefined behavior occurs.

I [suggested](#) a trick involving a volatile variable:

```
inline void log(const char *msg)
{std::fprintf(stderr, msg);} // always returns

bool on_fire() {
    static volatile bool fire; // always false
    return fire;
}

void f(int *p) {
    if (p == nullptr) log("bad thing 1");
    if (on_fire()) std::abort();
    if (*p >= 5) log("bad thing 2");
}
```

The idea is that the compiler cannot assume that `on_fire()` returns `false`, and so the check for `p` being null cannot be eliminated. However, the compiler can observe that, if `p` is null, the behavior will be undefined *unless* `on_fire()` returns `true`, and so it can elide that check (though not the volatile read) and call `abort()`. This therefore seems to convey a certain capability of *observing* the upcoming undefined behavior without actually experiencing it.

Unfortunately, conforming implementations are not constrained to follow this analysis. The volatile read is observable behavior, and it is logically necessary that the implementation perform that read unless it can somehow obtain its result otherwise. However, after reading the value `false` (as of course it will be in practice) the implementation may take any action whatsoever, even “undoing” the call to `log`. For

example, it would be permissible to perform the implicit flush for `stderr` only just before the call to `std::abort` (which never happens). One might hope for the implementation to allow for the possibility that `log` affects some hardware state that affects the volatile read, but it might not as such a scheme would require support from the operating system.

General solution

We can instead introduce a special library function

```
namespace std {
    // in <stdlib>
    void observable() noexcept;
}
```

that divides the program's execution into *epochs*, each of which has its own observable behavior. If any epoch completes without undefined behavior occurring, the implementation is required to exhibit the epoch's observable behavior. Ending an epoch is nonetheless distinct from ending the program: for example, there is no automatic flushing of `<stdio>` streams.

Undefined behavior in one epoch may obscure the observable behavior of a previous epoch (for example, by re-opening an output file), but external mechanisms such as pipes to a logging process can be used to guarantee receipt of an epoch's output. Normal thread synchronization is required for the observable behavior of one thread to be included in an epoch defined by another.

As a practical matter, a compiler can implement `std::observable` efficiently as an intrinsic that counts as a possible termination, which the optimizer thus cannot remove. After optimization (including any link-time optimization), the code generator can then produce zero machine instructions for it.

Limited assumptions

A call to `std::observable` prevents the propagation of assumptions based on the potential for undefined behavior after it into code before it. The following functions offer the same opportunities for optimization:

```
void a(int &r, int *p) {
    if (p) std::fprintf(stderr, "count: %d\n", ++r);
    if (!p) std::abort(); // henceforth, p is known to be non-null
    if (!p) std::fprintf(stderr, "p is null\n");
}

void b(int &r, int *p) {
    if (p) std::fprintf(stderr, "count: %d\n", ++r);
    std::observable();
    if (!p) std::fprintf(stderr, "p is null\n");
    *p += r; // p may be assumed non-null
}
```

In both cases, the “p is null” output can be elided: in a, because execution would not continue past the `std::abort`; in b, because of the following dereference of p. In both cases, the count output must appear if p is null: in a, because the program thereafter has the defined behavior of aborting; in b, because the epoch ends before undefined behavior occurs.

The function b, however, offers the additional optimization of not checking the condition at run time. We therefore achieve the controlled optimization benefits of a checked contract (without continuation) without the runtime efficiency of an unchecked contract.

Usage

Adding a call to `std::observable` into a contract condition guarantees that previous contracts are checked, even if the violation handler always returns:

```
void f(int *p) [[expects: p]] [[expects: (std::observable(), *p<5)]];
```

This is permitted because `std::observable()` has no side effects ([dcl.attr.contract.syn]/6). If the condition is not checked because of the build level (and it isn't evaluated anyway ([dcl.attr.contract.check]/4)), no epoch is ended but no undefined behavior occurs.

Using `std::observable` in a contract condition does not, however, contain the undefined behavior from the condition being (known to be) false when *not* checked, since it is unspecified whether the condition is evaluated ([dcl.attr.contract.check]/4) and thus whether an epoch boundary occurs. A call can instead be inserted *before* a call to a function to contain assumptions based on its preconditions:

```
void f(int i) [[expects: i>10]];
void client(int i) {
    if (i < 5) log("small input"); // will not be elided
    std::observable();
    f(i);
}
```

Such a call can be added to a wrapper for a function with a precondition or to a logging function to prevent the elision of any logging statement based on subsequent code. Added to a violation handler, it prevents the elision of any (enabled) contract check based on (unsurprising) undefined behavior following it.

Wording

Add a paragraph before [intro.abstract]/5:

An *observable checkpoint* is a call to `std::observable` ([support.start.term]) or program termination.

Change [intro.abstract]/5 as follows:

A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this document places no requirement on the implementation executing that program with that input ~~(not even with regard to operations preceding~~ with regard to any operation that does not happen before an observable checkpoint that happens before the first undefined operation).

Change [intro.abstract]/6.2 as follows:

~~At program termination~~ At each observable checkpoint, all data whose delivery to the host environment to be written ~~into files to~~ any file happens before that checkpoint shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced. [Note: Not all host environments provide access to file contents before program termination. — end note]

[Drafting note: The phrase “delivery to ... any file” refers to C11 7.21.5.2/2. — end note]

Add to [cstdlib.syn]:

```
[[noreturn]] void quick_exit(int status) noexcept;  
void observable() noexcept;
```

Add paragraphs to the end of [support.start.term]:

```
void observable() noexcept;
```

Effects: Establishes an observable checkpoint ([intro.abstract]). No other effects.